

SIMPLE PROGRAMMING

The 10 Minute Guide to Bitwise Operators

(Cause you've got 10 minutes until your interview starts and you know you should probably know this, right?)

Twitter: @FayeWilliams

Web: www.fayewilliams.com

Bitwise Operators

If you are lucky enough to work on low-level, embedded systems (like me!), then eventually, someday, you will probably have to make sense of bitwise manipulation, and when that day comes, you will just LOVE this guide, because it makes them super easy to understand.

And even if you think you won't use them, well, you could read this guide anyway, because you never know when a pub quiz is gonna ask about exclusive OR, and like the scouts, it's good to be prepared.

Let's get started

The bitwise operators enable you to work at a very fine-grained level, which is advantageous when you are dealing with limited space, or sending messages with limited capacity.

What the hell does that mean?

Ah, sorry about the textbook spiel there, let's start again.

Bitwise operators allow us to work at a sub-byte level. You know what a byte is – computer memory is made up of lots of them. Well, a byte itself is divided up into 8 bits.

Each one of these teeny, tiny bits has one of two states. It is either ON or OFF.

Usually, in computing, a zero represents OFF and a 1 represents ON. No, I don't know why I'm using capitals either.

A bit is the smallest piece of storage or data that a computer can utilise. You can't do anything with less than a bit.

Now, it turns out that you can't directly access these bits, but they *are* readable and writeable - if you know how to use the bitwise operators.

All clear so far?

Great.

In order to read and write individual bit values, C provides us with four operators:

- & Bitwise AND
- | Bitwise OR (inclusive OR)
- ^ Bitwise XOR (exclusive OR)
- ~ Bitwise NOT (or "one's complement")

What about bit shifting?

Yes, smarty-pants, there are also two bit-shift operators:

<< and >>

However, these don't operate on individual bits, they merely move all the bits left or right.

If you want to know more, there's a [post on my website that provides a practical example of bit shifting](#). I won't be covering it here.

Binary numbers

I use binary numbers to illustrate the concepts in this guide. If you need a reminder on how they work, [there's a short and sweet explanation here](#).

OK. Enough intro. Let's have a look at each of the bitwise operators in turn. We'll see how they work and talk through an example or two of what on earth you would use them for in 'real life'.

Bitwise AND: &

Bitwise AND is represented by the '&' symbol (ampersand).

Example:

$$128 \ \& \ 10 = 0$$

What is this actually doing?

Bitwise AND compares each bit setting and **if they match**, sets the corresponding output bit to 1. So using the example above, we can visualise the actual bit comparisons as follows:

$$\begin{array}{rcl} 10000000 & // & 128 \\ \& \ 00001010 & // \ 10 \\ \hline 00000000 & // & 0 \end{array}$$

What we're doing here is comparing the top bit and the bottom bit. If they are the same, the output is 1. If they are different the output is 0.

Crazy? Unintuitive? Glorious?

Yes, all of the above.

What would I use bitwise AND for?

Good question.

The above example seems totally abstract, and quite clearly would make you wonder what use this would ever actually have.

To help cement the concepts in your mind I'm going to provide examples that will allow you to see the beauty of these special operators. Seeing them in use will also help you understand their value programmatically.

1. Obtain the value of a bit

Let's say I am using a byte to hold the true/false status of 8 items. I want to know if item 5 is true (i.e. set to 1).

Now, you can't read bits individually - have I said that already?

You can read the contents of a byte, but you can't access individual bits without using the operators.

So, since I can't use something like `byte[4]` to access the 5th element, instead I'll AND my byte with the appropriate mask and test the result.

Whoa. Mask? What mask?

A mask is just a byte that is set to a fixed value. Since you know which bits are set in your mask, you can use it to check the status of an unknown byte.

When you AND two bytes together, the value of the mask is returned if the relevant bits match.

In this case, my mask is decimal 16 (or, 00010000). See how (reading right to left), I have the bit set to 1 at position 5? This is the value I am checking for:

```
01101011    // 107
& 00010000    // 16 (Mask)
-----
00000000    // 0
```

The result is zero, because **bit 5 is not set**.

Let's check a different byte:

```
01111011    // 123
& 00010000    // 16 (Mask)
-----
00010000    // 16
```

Now the result is 16, because **bit 5 is set**.

2. Turn off an individual bit

You can also use AND in 'reverse', and turn off selected bits using a fixed mask.

Let's say that I want to turn off bit 5. I can do this with the mask 239 (11101111). The result is the original input, but with bit 5 switched off:

```
    01111011    // 123
& 11101111    // 23 (Mask)
-----
    01101011    // 107
```

Can you see what's happened here?

The bitwise AND comparison checks each bit and sets the result to 1 if they match.

Therefore, if I want to turn off bit 5, all I need to do is set my mask to contain zero at this position. That way, bit 5 will never be set to 1 in the result.

Setting everything else to 1 in the mask ensures that **all other bits remain unchanged**.

Note that you can't turn a bit on with the AND operator, since you must have two values of 1 to get the output of 1 (but see bitwise OR for this).

Bitwise OR: |

Bitwise OR is represented by the 'l' symbol (pipe).

Example:

$130 \mid 10 = 138$

What is this actually doing?

Bitwise OR compares each bit setting and **if either one is set**, sets the corresponding output bit to 1. So in the example above, we can see the actual bit comparisons as follows:

10000010	// 130
00001010	// 10
<hr/> 10001010	// 138

Note that it doesn't matter if one or both bits are set in what we are comparing. As long as either one is set, the output bit is set to 1. This is what makes it *inclusive*.

What would I use bitwise OR for?

An example follows – it's much easier to understand the usefulness of these operators when you see them in action.

1. Turn on an individual bit

Remember we used bitwise AND to turn *off* an individual bit? Well bitwise OR allows you to do the opposite.

Let's say we have a byte, containing 8 bits, which are used as true/false values. Without changing the settings of any of the other bits, we want to set the first bit to be true. We apply the mask 1 (decimal) as follows:

```
    11000100    // 196
| 00000001    // 1 (Mask)
-----
    11000101    // 197
```

The result is that the first bit (that is, the right-most bit) is now set to 1, and the other bit settings remain unchanged.

That's all there is to it!

Bitwise XOR: ^

Bitwise XOR is represented by the '^' symbol (caret or 'hat').

Example:

$$130 \wedge 10 = 136$$

What is this actually doing?

Bitwise XOR is an *exclusive* OR operation – it compares each bit setting and **if either one is set**, sets the corresponding output bit to 1. However, **if both bits are set**, then the corresponding output bit is set to zero.

So to clarify, both OR and XOR compare each bit setting and if either one is set, sets the corresponding output bit to 1. However, if both bits are set, OR sets the output to 1 (inclusive), while XOR sets the output to zero (exclusive).

In short, XOR gives a result of 1 only if the two inputs are different.

So the example above has bit comparisons as follows:

```
    10000010    // 130
^   00001010    // 10
-----
    10001000    // 136
```

What would I use bitwise XOR for?

Let's take a look at how we might use XOR.

1. Toggle bits

XOR is primarily used to toggle bits on and off. Let's say we have a byte which represents 8 boolean values and we want to turn the first two off and on again.

We can use the same mask to toggle the bits we are interested in:

```
    10000111    // 135
^   00000011    // 3  (Mask)
-----
    10000100    // 132
```

Applying the mask turns off the last two bits.

```
    10000100    // 132
^  00000011    // 3  (Mask)

    10000111    // 135
```

Applying it again (to the result from above) turns the bits back on.

All the other bits remain unchanged. XOR is just like applying a switch – very useful in low level programming.

Bitwise NOT ~

Bitwise NOT is represented by the '~' symbol (tilde).

Example:

$\sim 125 = 130$

Bitwise NOT only takes one operand. This means that it is a **unary** operator.

What is this actually doing?

Bitwise NOT essentially 'flips' (i.e. changes) the bit setting for each and every bit, resulting in what is known as the *one's complement* of the original number.

So the example above has a bit output as follows:

~ 01111101	// 125
<hr/>	
10000010	// 130

What would I use bitwise NOT for?

1. Turn off an individual bit

Actually, the NOT operator doesn't itself turn off a bit, but in conjunction with the AND operator, you can use it with a mask to turn off a bit in a nice readable manner.

This is best illustrated with a full example:

```
char settings = 0;      // 00000000
char FLAG1 = 1;        // 00000001
```

Let's assume that *settings* is a byte we use to represent 8 boolean values. First of all, we'll set the first bit of our settings variable to 1 using the OR operator and our FLAG1 mask:

```
settings = settings | FLAG1;
```

In binary this is:

```
    00000000    // 0 (settings)
|  00000001    // 1 (FLAG1)
-----
    00000001    // 1 (settings)
```

Now, if we want to turn that bit off again, instead of using AND and defining a new mask (11111110), we can use our existing FLAG1 mask and the NOT operator as follows:

```
settings = settings & ~FLAG1;
```

This reads quite intuitively: *settings AND NOT FLAG1* turns off the first bit in settings. In binary:

```
00000001    // 1 (settings)
& 11111110    // 254 (one's complement of
                FLAG1)
-----
00000000    // 0 (settings - back to our
                original number)
```

Note that applying the NOT operator to the flag generates the same binary sequence we would need to define as a new mask if we wanted to use AND alone (11111110).

The NOT operator saves us a variable (important where you have space constraints).

Can't I just use XOR to turn this bit on and off?

Well spotted, yes you can!

```
//turn first bit on or off  
settings = settings ^ FLAG1;
```

The difference with XOR is:

1. You may not know if you are turning the bit on or off without checking the result (or input) as it's just a switch (which means more instructions)
2. XOR always has an effect, whether you intend it to or not.

With & ^ (AND NOT) you know you are explicitly turning a bit off, and **if the bit is already off when you apply the mask, the result is unchanged.**

Awesome eh?

Bitwise operator quick start guide

In summary, which operator to use when.

& Bitwise AND

- Use with a mask to check if bits are on or off
- Turn off individual bits

| Bitwise OR

- Turn on individual bits

^ Bitwise XOR

- Toggle bits on and off, like a switch

~ Bitwise NOT

- Turn off individual bits with AND